Spring, 1992 Name: _____

CS 342 — Principles of Programming Languages

LISP Test (Problem Solving, Version 1)

Before you begin, please take a moment to look over the entire test so that you can budget your time. For programming problems, there is usually more than one right answer, hence it is important to present with your program evidence that it is correct. Answers to programming questions will be scored on the following points:

- elegance and clarity (conciseness, using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution, and how clear and convincing your argument is that the code is correct)
- correctness (whether your program solves the given problem correctly in the given programming language)

You should try to eliminate all syntax errors from your programs, as they will affect both the correctness and clarity of your solution; you may receive no points for programs with major syntax errors. Run-time space and time efficiency of programs is not important, unless the problem says so. Also, checking of external inputs, prompting, labeling output, and other conventions for dealing with humans are not important (for this class), unless the problem says so.
The breakdown of points between these categories is roughly as follows.

- 60% for elegance and clarity (using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution),
- 40% for correctness (whether your program solves the given problem correctly in the given subset of the given language)

You are encouraged to use auxiliary subroutines whenever they are helpful.

You may use any of the following functions from the book in your solutions. Any other procedures that you wish to use must be defined in your solution. If you wish to use functions from your solutions in other parts of the exam, you must clearly state where we can find those function definitions.

```
(define length (l) (if (null? l) 0 (+1 (length (cdr l)))))
(define caar (l) (car (car l)))
(define cadr (l) (car (cdr l)))
(define cddr (l) (cdr (cdr l)))
(define caddr (l) (car (cdr (cdr l))))
(define cadar (l) (car (cdr (car l))))
(define cadddr (exp) (car (cdr (cdr (cdr exp)))))
(define list1 (x) (cons x '()))
(define list2 (x y) (cons x (cons y '())))
(define list3 (x y z) (cons x (cons y (cons z '()))))
(define or (x y) (if x x y))
(define and (x y) (if x y x)).
(define not (x) (if x '() 'T)).
(define atom? (x) (or (null? x) (or (number? x) (symbol? x))))
(define equal (l1 l2)
      (if (atom? l1) (= l1 l2)
          (if (atom? l2) '()
              (if (equal (car l1) (car l2))
                   (equal (cdr l1) (cdr l2))
                   '()))))
(define assoc (x alist)
     (if (null? alist) '()
         (if  (= x (caar alist)) (cadar alist)
             (assoc x (cdr alist)))))
(define mkassoc (x y alist)
     (if (null? alist)
          (list1 (list2 x y))
         (if  (= x (caar alist)) (cons (list2 x y) (cdr alist))
           (cons (car alist) (mkassoc x y (cdr alist))))))
(define addelt (x s) (if (member? x s) s (cons x s)))
(define member? (x s)
     (if (null? s) '()
         (if (equal x (car s)) 'T (member? x (cdr s)))))
(define size (s) (length s))
(define union (s1 s2)
     (if (null? s1) s2
         (if (member? (car s1) s2)
                (union (cdr s1) s2)
             (cons (car s1) (union (cdr s1) s2)))))
(define inter (s1 s2)
     (if (null? s1) s1
          (if (member? (car s1) s2)
               (cons (car s1) (inter (cdr s1) s2))
               (inter (cdr s1) s2))))
(define diff (s1 s2)
     (if (null? s1) s1
         (if (null? s2) s1
              (if (member? (car s1) s2)
                   (diff (cdr s1) s2)
                   (cons (car s1) (diff (cdr s1) s2))))))
(define append (x y) (if (null? x) y (cons (car x) (append (cdr x) y))))
```

1 Problem Solving

1. Write a function `(snoc lst x)`, that takes a list `lst` and a s-expression `x` and returns a list with the original elements plus `x` added to the end of `lst`; that is, such that `(rac (snoc` l x`))` $= x$. (Note: `snoc` is `cons` spelled backwards.) For example, `(snoc '(a b c) 'd)` = `(a b c d)`.

2. Write a function `(first-n num lst)` that takes a non-negative number `num`, a list `lst` and returns a list consisting of the first `num` elements of `lst` in the same order. You may assume that `lst` has at least `num` elements. For example: `(first-n 3 '(a b c d e))` returns `(a b c)`. For example: `(first-n 0 '(a b c d e))` returns `()`.

3. Write a function `(rac lst)` that returns the last element of a non-empty list. (Note: `rac` is `car` spelled backwards.) For example, `(rac '(a b c))` = `c`. For efficiency, in this problem (only), your solution may *not* call `cons`, either directly or indirectly.

4. Consider the following LISP code:

```
(set the-empty-tree '(tree))
(define tree-make (left node right)   ; : tree, s-exp, tree -> tree
  (cons 'tree (cons left (cons node (cons right '()))))))
(define leaf-make (node)              ; : s-exp -> tree
  (tree-make the-empty-tree node the-empty-tree))
```

These represent binary trees of s-expressions; for example `(leaf-make 3)` returns the list `(tree (tree) 3 (tree))` which represents a tree with root 3 and empty subtrees; similarly, `(tree-make t1 27 t2)` represents a tree whose left subtree is the value of `t1`, whose right subtree is the value of `t2`, and whose root holds the value 27. The root of a tree and all of the roots of its subtrees are called nodes.

(a) Write a function `(post-order tree)` returns a list containing the node values of the tree `tree` with the order given by a post-order traversal. That is, `(post-order the-empty-tree)` returns `()`, and `(post-order (tree-make` t_1 x t_2`))` returns the list containing first the elements of `(post-order` t_1`)`, the elements of `(post-order` t_2`)`, and then x. For example, `(post-order (tree-make (leaf-make 1) 3 (leaf-make 2)))` returns the list `(1 2 3)`.

(b) Write a function `(same-set t1 t2)` that takes two trees as arguments and returns true if `t1` and `t2` have the same set of elements. For example, `(same-set the-empty-tree the-empty-tree)` is true, and so is

```
(same-set (tree-make (leaf-make 3) 4 (leaf-make 3))
          (tree-make (leaf-make 4) 3 the-empty-tree))
```